

# Accelerating weather models with PGI compilers

The Portland Group

[www.pgroup.com](http://www.pgroup.com)

dave.norton@pgroup.com

# PGI

# CUDA Fortran in 3 slides

# CUDA Fortran VADD Host Code

```
subroutine vadd( A, B, C )  
  use cudafor  
  use kmod  
  real, dimension(:) :: A, B  
  real, pinned, dimension(:) :: C  
  real, device, allocatable:: Ad(:), Bd(:), Cd(:)  
  integer :: N  
  N = size( A, 1 )  
  allocate( Ad(N), Bd(N), Cd(N) )  
  Ad = A(1:N)  
  Bd = B(1:N)  
  call vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )  
  C = Cd  
  deallocate( Ad, Bd, Cd )  
end subroutine
```

# CUDA Fortran VADD Device Code

```
module kmod
  use cudafor
contains
  attributes(global) subroutine vaddkernel(A,B,C,N)
    real, device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine
end module
```

# Building a CUDA Fortran Program

- CUDA Fortran is supported by the PGI Fortran compilers when the filename uses a CUDA Fortran extension. The `.cuf` extension specifies that the file is a free-format CUDA Fortran program;
- The `.CUF` extension may also be used, in which case the program is processed by the preprocessor before being compiled.
- To compile a fixed-format program, add the command line option `-Mfixed`.
- CUDA Fortran extensions can be enabled in any Fortran source file by adding the `-Mcuda` command line option.
- Most F2003 features should work in CUDA Fortran.
- There is a (CUDA-like) API to access features
  - Streams supported through API rather than language

Accelerator Directives for flat  
performance profile codes  
in 6 slides

# Accelerator VADD Device Code

## (two dimensional array example)

```
module kmod
  contains
  subroutine vaddkernel(A,B,C)
    real :: A(:, :), B(:, :), C(:, :)
    !$acc region
      C(:, :) = A(:, :) + B(:, :)
      <lots of other code to do neat stuff>
      <special code to do even neater stuff>
    !$acc end region
  end subroutine
end module
```

**!\$acc region clauses can surround many individual loops and compute kernels. There is no implicit GPU/CPU data movement within a region**

# Compiling the subroutine:

```
PGI$ pgfortran -Minfo=accel -ta=nvidia -c vadd.F90
```

vaddkernel:

5, Generating copyout(c(1:z\_b\_14,1:z\_b\_17))

Generating copyin(a(1:z\_b\_14,1:z\_b\_17))

Generating copyin(b(1:z\_b\_14,1:z\_b\_17))

Generating compute capability 1.0 binary

Generating compute capability 1.3 binary

Generating compute capability 2.0 binary

6, Loop is parallelizable

Accelerator kernel generated

6, !\$acc do parallel, vector(16) ! blockidx%x threadidx%x

!\$acc do parallel, vector(16) ! blockidx%y threadidx%y

CC 1.0 : 7 registers; 64 shared, 8 constant, 0 local memory bytes; 100% occupancy

CC 1.3 : 8 registers; 64 shared, 8 constant, 0 local memory bytes; 100% occupancy

CC 2.0 : 15 registers; 8 shared, 72 constant, 0 local memory bytes; 100% occupancy



# Tuning the compute kernel

## Accelerator VADD Device Code

```
module kmod
  contains
  subroutine vaddkernel(A,B,C)      ! We know array size
    real :: A(:, :), B(:, :), C(:, :) ! dimension(2560,96)
    integer :: i,j
    !$acc region
    !$acc do parallel
      do j = 1,size(A,1)
        !$acc do vector(96)
          do i = 1,size(A,2)
            C(j,i) = A(j,i) + B (j,i)
          enddo
        enddo
      enddo
    !$acc end region
  end subroutine
end module
```

# Keeping the data on the GPU

## Accelerator VADD Device Code

```
module kmod
  contains
  subroutine vaddkernel(A,B,C)
    real :: A(:, :), B(:, :), C(:, :)
    !$acc reflected (A,B,C)
    !$acc region
      C(:, :) = A(:, :) + B(:, :)
    !$acc end region
  end subroutine
end module
```

**The !\$reflected clause must be visible to the caller so it knows to pass pointers to arrays on the GPU rather than copyin actual array data.**

# Compiling the subroutine:

```
PGI$ pgfortran -Minfo=accel -ta=nvidia -c vadd.F90
```

```
vaddkernel:
```

```
5, Generating reflected(c(:,:))
```

```
Generating reflected(b(:,:))
```

```
Generating reflected(a(:,:))
```

```
6, Generating compute capability 1.0 binary
```

```
Generating compute capability 1.3 binary
```

```
Generating compute capability 2.0 binary
```

```
7, Loop is parallelizable
```

```
Accelerator kernel generated
```

```
7, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
```

```
!$acc do parallel, vector(16) ! blockidx%y threadidx%y
```

```
CC 1.0 : 11 registers; 80 shared, 8 constant, 0 local memory bytes; 66% occupancy
```

```
CC 1.3 : 11 registers; 80 shared, 8 constant, 0 local memory bytes; 100% occupancy
```

```
CC 2.0 : 17 registers; 8 shared, 88 constant, 0 local memory bytes; 100% occupancy
```

# Allocating/Deallocating GPU Arrays

## Accelerator VADD Device Code

```
subroutine vadd(M,N,C)
  use kmod ! Visibility of !$acc reflected
  real, dimension(:,:) :: A, B, C
  integer :: N
  !$acc mirror(A,B)    !device resident clause in 1.3
  allocate(A(M,N),B(M,N))
  ! C has been mirrored and allocated previously
  A = 1.0
  B = 2.0
  !$acc update device(A,B,C)
  call vaddkernel (A,B,C)
  call kernel2 (A,B,C)
  call kernel3 (A,B,C)
  call kernel4 (A,B,C)
  !$acc update host(C)
  deallocate( A, B)
end subroutine
```

# Using GPU device-resident data across subroutines

```
subroutine timestep(Input,Result,M,N)
  use kmod ! Make reflected var's visible
  real, dimension(M,N) :: Input,Result

  integer :: M,N
  real, allocatable :: B,C,D
  dimension(:,:) :: B,C,D
  !$acc mirror(B,C,D)
  allocate(B(M,N),C(M,N),D(M,N))
  B = 2.0
  !$acc update device(Input,B)
  call vaddkernel(Input,B,C)
  ...
  call kernel2(C,D)
  ...
  call kernel3(D,Result)
  !$acc update host(Result)
  deallocate(B,C,D)
end subroutine
```

CPU Code

```
module kmod
  Contains
  !
  subroutine vaddkernel(A,B,C)
    real :: A(:,:),B(:,:),C(:,:)
    !$acc reflected(A,B,C)
    !$acc region
    C(:,:) = A(:,:) + B(:,:)
    !$acc end region
  end subroutine
  !
  subroutine kernel2(C,D)
    real :: C(:,:),D(:,:)
    !$acc reflected(C,D)
    !$acc region
    < compute-intensive loops >
    !$acc end region
  end subroutine
  ...
end module
```

GPU Code

# CUDA-x86

PGI®

engadget 



```
% pgfortran -help -ta
```

```
-ta=nvidia:{analysis|nofma|[no]flushz|keepbin|keepptx|keepgpu|maxregcount:<n>|  
          c10|cc11|cc12|cc13|cc20|fastmath|mul24|time|cuda2.3|cuda3.0|  
          cuda3.1|cuda3.2|cuda4.0|[no]wait}|host
```

Choose target accelerator

nvidia Select NVIDIA accelerator target

analysis Analysis only, no code generation

nofma Don't generate fused mul-add instructions

[no]flushz Enable flush-to-zero mode on the GPU

keepbin Keep kernel .bin files

keepptx Keep kernel .ptx files

keepgpu Keep kernel source files

maxregcount:<n> Set maximum number of registers to use on the GPU

cc10 Compile for compute capability 1.0

...

cc20 Compile for compute capability 2.0

fastmath Use fast math library

mul24 Use 24-bit multiplication for subscripting

time Collect simple timing information

cuda2.3 Use CUDA 2.3 Toolkit compatibility

...

cuda4.0 Use CUDA 4.0 Toolkit compatibility

[no]wait Wait for each kernel to finish; overrides nowait clause

host Compile for the host, i.e. no accelerator target

# Compute region directive clauses for tuning data allocation and movement

Clause	Meaning
<code>if (<i>condition</i>)</code>	Execute on GPU conditionally
<code>copy (<i>list</i>)</code>	Copy in and out of GPU memory
<code>copyin (<i>list</i>)</code>	Only copy in to GPU memory
<code>copyout (<i>list</i>)</code>	Only copy out of GPU memory
<code>local (<i>list</i>)</code>	Allocate locally on GPU
<code>deviceptr (<i>list</i>)</code>	C pointers in <i>list</i> are device pointers
<code>update device (<i>list</i>)</code>	Update device copies of the arrays
<code>update host (<i>list</i>)</code>	Update host copies of the arrays



# Loop directive clauses for tuning GPU kernel schedules

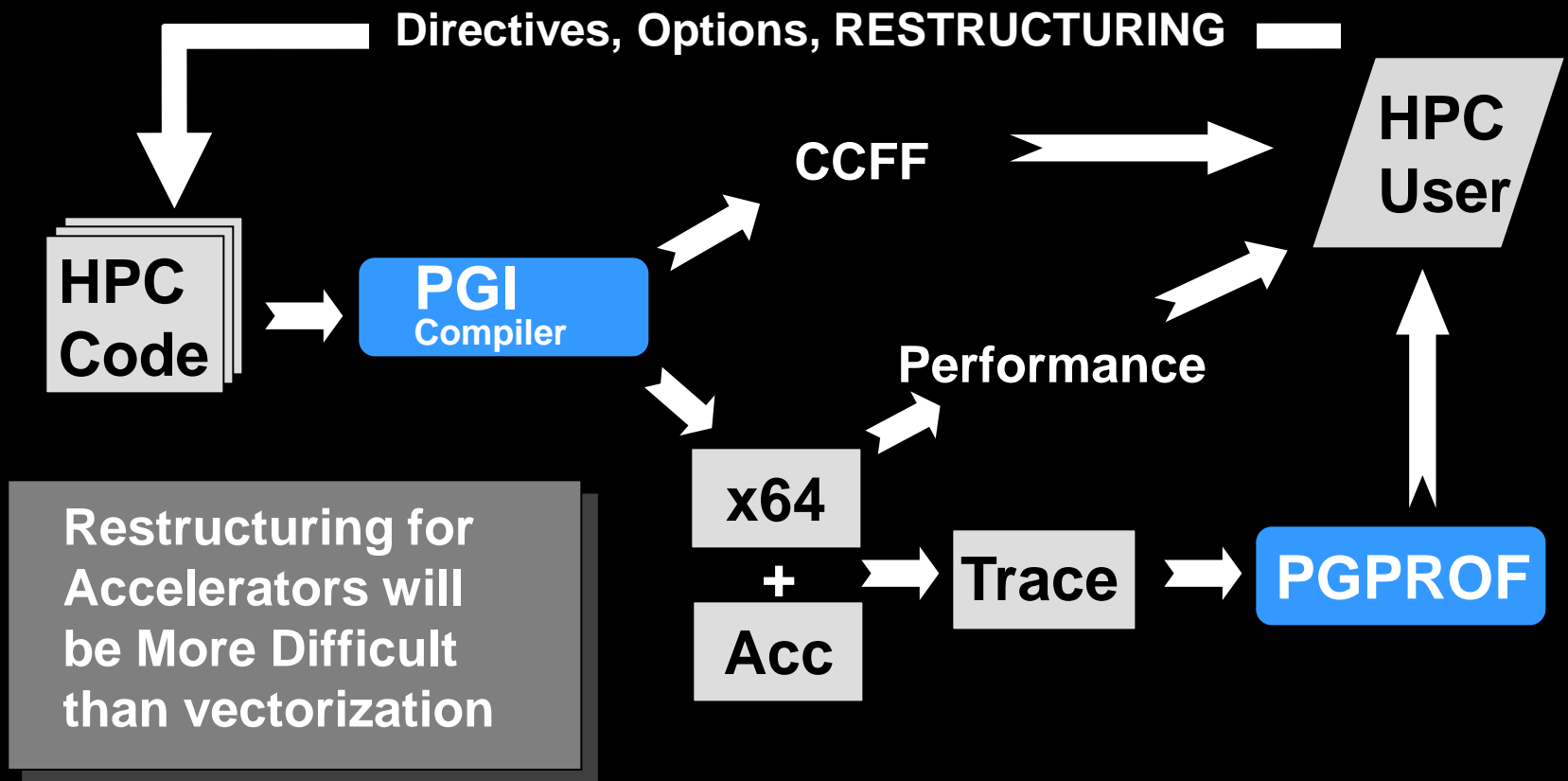
Clause	Meaning
<code>parallel [(width)]</code>	Parallelize the loop across the multi-processors
<code>vector [(width)]</code>	SIMD vectorize the loop within a multi-processor
<code>seq [(width)]</code>	Execute the loop sequentially on each thread processor
<code>independent</code>	Iterations of this loop are data independent of each other
<code>unroll (factor)</code>	Unroll the loop <i>factor</i> times
<code>cache (list)</code>	Try to place these variables in shared memory
<code>private (list)</code>	Allocate a copy of each variable in <i>list</i> for each loop iteration

# Timing / Profiling

- How long does my program take to run?
  - `time ./myprogram`
- How long do my kernels take to run?
  - `pgfortran -ta=nvidia,time`
- Environment variables:
  - `export ACC_NOTIFY=1`
  - `export NVDEBUG=1`
  - `# cuda profiler settings`
  - `#export CUDA_PROFILE=1`
  - `#export CUDA_PROFILE_CONFIG=cudaprof.cfg`
  - `#export CUDA_PROFILE_CSV=1`
  - `#export CUDA_PROFILE_LOG=cudaprof.log`

# Compiler-to-Programmer Feedback

Incremental porting/tuning for GPUs



# Obstacles to GPU code generation

- Loop nests to be offloaded to the GPU must be rectangular
- At least some of the loops to be offloaded must be fully data parallel with no synchronization or dependences across iterations
- Computed array indices should be avoided
- All function calls must be inlined within loops to be offloaded
- In Fortran, the pointer attribute is not supported; pointer arrays may be specified, but pointer association is not preserved in GPU device memory
- In C
  - Loops that operate on structs can be offloaded, but those that operate on nested structs cannot
  - Pointers used to access arrays in loops to be offloaded must be declared with C99 restrict (or compiled w/-Msafe\_ptr, but it is file scope)
  - Pointer arithmetic is not allowed within loops to be offloaded

# Evolution of the Directives


- Published Version 1.0 of the PGI Accelerator Directives
  - Intent of publication was to start discussion on standardization process
- Implemented v1.0
- Standardization process started through OMP
- Published Version 1.3 of the PGI Accelerator Directives
- Currently implementing v1.3

# PGI

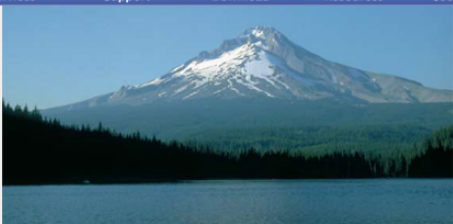
- C99, C++, F2003 Compilers
  - Optimizing
  - Vectorizing
  - Parallelizing
- Graphical parallel tools
  - PGDBG® debugger
  - PGPROF® profiler
- AMD, Intel, NVIDIA, ST
- 64-bit / 32-bit
- PGI Unified Binary™
- Linux, MacOS, Windows
- Visual Studio integration
- GPGPU Features
  - CUDA Fortran
  - PGI Accelerator™
  - CUDA-x86

## The Portland Group

[Technology](#)[Products](#)[Services](#)[Support](#)[Download](#)[Resources](#)[User Forums](#)[Purchase](#)[About](#)




**PGI Accelerator Files**  
Articles, tutorials, source code and benchmarks to help you with your x64+GPU software development.




## PGI 2011

is now available for [download](#). This release includes full support for Fortran 2003, full support for the PGI Accelerator Programming Model v1.2, significant C++ performance improvements and more. Read [what's new](#).


### PGI® Optimizing Fortran, C and C++ Compilers & Tools




**PGI Workstation™ and PGI Server™ for x64**  
PGI optimizing multi-core x64 compilers for Linux, MacOS & Windows with support for debugging and profiling of local MPI processes. A complete OpenMP/MPI SDK for high performance computing on the latest Intel and AMD CPUs. [More info](#) | [Try](#) | [Buy](#)




**CUDA Fortran**  
CUDA Fortran enables GPU acceleration of HPC applications using the NVIDIA CUDA parallel programming model in a native optimizing Fortran 2003 compiler. Compatible and interoperable with NVIDIA's C for CUDA. [More info](#) | [Try](#) | [Buy](#)



**PGI Accelerator™ C99 & Fortran**  
PGI Accelerator C99 & Fortran enable high level programming of HPC applications for x64+GPU platforms using OpenMP-like compiler directives. Portable, incremental, and easy to use for application domain experts. [More info](#) | [Try](#) | [Buy](#)



**The PGI CDK® Cluster Development Kit®**  
The PGI CDK includes optimizing Fortran/C/C++ compilers configured to build, debug and profile MPI and hybrid MPI/OpenMP HPC applications for Linux or Windows Clusters using the major open source MPI implementations or MSMPI. [More info](#) | [Try](#) | [Buy](#)



**PGI Visual Fortran® for Microsoft Windows**  
PGI Visual Fortran brings optimizing multi-core x64 Fortran with integrated OpenMP/MPI debugging to scientists & engineers on Microsoft Windows within Microsoft Visual Studio. [More info](#) | [Try](#) | [Buy](#)

#### The PGI Accelerator Programming Model on NVIDIA GPUs Part 3: Porting WRF

Part 3 in the series looks at porting a key module in the Weather Research and Forecasting (WRF) application to GPUs.

#### Accessing Compiler Performance Advice

Using PGI compiler feedback with the PGPROF profiler can ease the task of improving application performance.

#### Using Microsoft MPI with PGI Workstation

Building, running and debugging MPI applications on Windows laptops and clusters using PGI Workstation.

#### Porting WRF to Microsoft Windows With PGI Workstation

A step by step guide to building the Weather Research and Forecasting application on Microsoft Windows using PGI Workstation.

#### The PGI Accelerator Programming Model on NVIDIA GPUs Part 4: New Features

Part 4 in the series looks at features in the PGI Accelerator compiler including support for leaving data on the GPU between kernels and support for GPU resident reductions.

#### Using PGPROF and the CUDA Visual Profiler to Profile GPU Applications

Tools and techniques for profiling both PGI Accelerator and CUDA Fortran programs.

#### CUDA Fortran Data Management

An overview and example of managing both CPU and GPU data using CUDA Fortran.

#### Tuning a Monte Carlo Algorithm on GPUs

A step-by-step example demonstrating many useful CUDA Fortran techniques including

Optimizing Performance

Installation

Buying PGI Products

[www.pgroup.com](http://www.pgroup.com)

# Reference Materials

- **PGI Accelerator programming model**
  - [http://www.pgroup.com/lit/whitepapers/pgi\\_accel\\_prog\\_model\\_1.3.pdf](http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf)
- **CUDA Fortran**
  - <http://www.pgroup.com/lit/whitepapers/pgicudaforug.pdf>
- **CUDA-x86**
  - <http://www.pgroup.com/resources/cuda-x86.htm>
- **Understanding the CUDA Threading Model**
  - <http://www.pgroup.com/lit/articles/insider/v2n1a5.htm>